The Passing of Time: The Magic Bakery

version 1.0

Typos/errors? Email sarah.clinch@manchester.ac.uk and I'll get them fixed.

1. Introduction

This is a supplementary document designed to make it clear how to implement the passing of time in Coursework II (The Magic Bakery). This document should be used in conjunction with the Coursework Manual, specification and rules PDF.

We suggest you don't read this document until you're ready to implement the `Customers` class.

2. Key methods for the passing of time

At the end of each round of the game, time passes. The rules explain how this works (page 6), but this is realised in our implementation through three separate methods in the `Customers` class:

- The `customerWillLeaveSoon` method indicates whether, given the current set of `activeCustomers`, a customer will leave at the end of the round
- The `timePasses` method moves `activeCustomers` through the shop in accordance with the rules, ensuring that a space becomes available in the *"leftmost"* space of the "*customer row*". If this results in a customer leaving the shop, then that `CustomerOrder` is returned by the `timePasses` method, if not then the `timePasses` method returns `null`.
- The `addCustomerOrder` method calls `timePasses` to create space in the shop. Then, if there are one or more cards in the `customerDeck`, the topmost card of the `customerDeck` is drawn and placed in the *"leftmost"* space of the "*customer row*" (i.e. they are added to `activeCustomers`).

3. Understanding the worked examples

Section 4 provides a worked example of the correct behaviour for the above three methods, given various states of `customerDeck` and `activeCustomers`.

The topmost set of examples (Block #1, rows 1-8), represent the behaviour whilst there are still `CustomerOrder`s remaining in the `customerDeck`. The bottommost set of examples (Block #2, rows 9-16) represent correct behaviour when there are no more `CustomerOrder`s remaining in the `customerDeck`.

Each row (numbered 1-16) depicts a single worked example. The two columns show the state of variables in the `Customers` class before a call to `addCustomerOrder` (left) and after the call to `addCustomerOrder` (right). Each of the small bordered boxes represents an exemplar `CustomerOrder` (i.e., Customer A, Customer B, Customer C and Customer D), where grey boxes indicate that the deck/space is empty/ `null`.

The state of three variables within the `Customers` class is shown in the five small bordered boxes in each column:

- The leftmost box shows the next card to be drawn from `customerDeck`.
- The middle three boxes represent the three spaces inside the shop, i.e. the `activeCustomers` variable. The arrows depict the direction of travel of customers within the shop (i.e. left to right), and `activeCustomers` therefore maintains a FIFO structure (i.e., the customer who least recently entered the shop will always be the one who moves to `inactiveCustomers`).
- The rightmost box shows the card that was most recently added to `inactiveCustomers`.

The coloured background of each `CustomerOrder` indicates that:

- White: the position of the `CustomerOrder` is unchanged between the before and after state,
- Yellow: the `CustomerOrder` has moved into/within `activeCustomers`
- Red: the `CustomerOrder` has become inactive

4. Worked examples

The worked examples are shown on the following page.


Examples Block #1 (Rows 1-8): There are CustomerOrders remaining in the customerDeck.
Before State
After State
Row #
customerDeck
activeCustomers
inactiveCustomers
1 D C B A
2 D C B
3 D C A
4 D B A
5 D C
6 D B
7 D A
8 D
Examples Block #2 (Rows 9-16): There are no more CustomerOrders remaining in the customerDeck.
Before State
After State
9 C B A
10 C B
11 C A
12 B A
13 C
14 B
15 A
16
Before State
After State